APPLICATION

FOR

UNITED STATES PATENT

TITLE: **SEGMENTED GLOBAL AREA DATABASE**

INVENTOR: STEPHEN K. MCGROGAN

ASSIGNEE: ADVENT SOFTWARE, INC.

ATTORNEY: STEPHEN C. GLAZIER
Kirkpatrick & Lockhart LLP
1800 Massachusetts Avenue, N.W.
Washington, DC 20036
202-778-9000
Reg. No. 31,361

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. _____ ET801840096US _____

I hereby certify under 37 CFR §1.10 that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner of Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

November 14, 2003
Date of Deposit

Carlo Cotrone (signature)
Signature

Carlo Cotrone
Typed or Printed Name of Person Signing Certificate

SEGMENTED GLOBAL AREA DATABASE

BACKGROUND

1. Field

[0001] Embodiments of the present invention relate generally to object-oriented databases (OODBs). More specifically, embodiments relate to memory management methods and systems in such databases.

2. Description of Related Art

[0002] In the information age, databases are a precious commodity, storing immense quantities of data for use in various applications. Latency, or time needed to access stored database data, is a crucial metric for many performance-intensive applications. Portfolio management applications, for example, are generally performance-intensive.

[0003] In-memory databases are the fastest possible databases. In such databases, which place the dataset in main memory, any piece of information is available with almost zero latency. The memory requirements of such databases increase with the size of the stored dataset. Therefore, such databases become excessively expensive from a hardware perspective when datasets are very large. In addition, computer manufacturers limit the amount of memory that can be installed in their machines, which limits the maximum size of the dataset that can be stored.

[0004] Some database systems address this memory problem by using software to cache portions of the dataset in main memory while keeping the majority in secondary memory (i.e., secondary storage), such as on disk. While this approach solves one problem, it creates another: Complex software must keep track of the location of the objects being stored, moving copies of the in-memory objects back and forth from the disk. This approach also increases complexity

2

and latency, as software must determine where to look for the object—in memory or on disk. In addition, desired data must be copied to the application's memory space because, for data integrity and functional reasons, users cannot be allowed direct access to the database copy of the object, whether it is found in the memory cache or on the disk.

SUMMARY

[0005] Embodiments of the present invention relate to database systems and methods.

[0006] In an embodiment, for the first time, data of a database is stored exclusively in secondary storage of a computer, not in main memory in the CPU. Instead, data is transparently mapped into and out of the main memory—not copied into the main memory—in response to reference patterns of an application program. Because with mapping, the data can be directly accessed by an application program at speeds close to those achievable if the data resided in the memory space of the application program, no copying of data need occur between secondary storage and main memory. As such, objects can be read by applications directly out of secondary storage with near zero latency, and without the database size restrictions of existing systems that copy database data into main memory.

[0007] In an embodiment, memory interrupt and virtual memory mapping facilities of computer hardware may be employed to make data appear to be in main memory when it actually resides in disk files on disk. That is, data can be accessed by applications directly out of secondary storage at speeds closely approximating existing systems that copy data into main CPU memory. No complex software is required to determine the residency of the data objects. The database may have an associated small fault or interrupt handler. If an object referenced by an application is not currently mapped into memory, the computer hardware, not software, will detect the fault. Then, the fault handler of the present invention will transparently map the appropriate disk file address into memory. Since the manufacturers of modern computers rely

on virtual mapping hardware and page swapping for overall machine speed, use of such facilities to map database data results in the fastest possible performance for the application.

[0008] Embodiments herein can greatly reduce the amount of database data that needs to reside in main CPU memory at any one moment. Accordingly, embodiments enable scalability to far larger datasets in secondary storage than previously possible, as well as the use of smaller, less expensive systems to perform current processing requirements. TCO (Total Cost of Ownership) is thus reduced. Database startup time is also greatly reduced, for data is placed in main memory only as needed, rather than loading all the data into memory before any processing can occur. Users also can reliably log into the database system even during periods of high volume transaction loading.

[0009] Embodiments herein may be used in connection with applications that interact with databases, such as investment portfolio management applications, for example.

[0010] In an embodiment, a database may be structured as a plurality of memory-mapped file segments stored on at least one nonvolatile memory medium. The file segments may include objects that are directly interconnected by memory pointers into one large matrix of information.

BRIEF DESCRIPTION OF THE DRAWINGS

[0011] FIG. 1 shows a hardware and software schema according to an embodiment of the present invention.

[0012] FIG. 2 shows a system according to an embodiment of the present invention.

[0013] FIG. 3 shows a database structure according to an embodiment of the present invention.

[0014] FIG. 4 shows a segmented data repository according to an embodiment of the present invention.

[0015] FIG. 5 shows a representation of memory-mapped segment files according to an embodiment of the present invention.

[0016] FIG. 6 shows a process according to an embodiment of the present invention.

[0017] FIG. 7A shows a process according to an embodiment of the present invention.

[0018] FIG. 7B shows a process according to an embodiment of the present invention.

[0019] FIG. 8 shows a process according to an embodiment of the present invention.

DETAILED DESCRIPTION

[0020] The following description refers to the accompanying drawings that illustrate certain embodiments of the present invention. Other embodiments are possible and modifications may be made to the embodiments without departing from the spirit and scope of the invention. Therefore, the following detailed description is not meant to limit the present invention. Rather, the scope of the present invention is defined by the appended claims.

[0021] Embodiments of the present invention relate to memory management methods and systems for databases, such as object-oriented databases (OODB).

[0022] In an embodiment, a database data repository includes a plurality of memory-mapped file segments stored on at least one nonvolatile memory medium. An application program connects to the data repository. A fault handler associated with the data repository is registered with the operating system of the application program. The fault handler catches a segmentation fault that is issued for an object referenced by the application program and resident in the data repository. A file segment corresponding to the referenced object is found and mapped into main memory. The application program is restarted at the interrupt location at which the segmentation fault was issued. Because data is transparently mapped into and out of the main memory without copying the data, objects may be read with near zero latency, and size restrictions on the database may be eliminated.

5

[0023] Although various embodiments herein are discussed in connection with portfolio management systems, it is to be appreciated that the present teachings may be implemented in any context which utilizes databases, such as, for example, a trade order management system (TOMS) or partnership accounting system.

[0024] Various embodiments herein have been implemented by Advent Software, Inc. (San Francisco, California) as the Geneva Segmented Advent Global Area (SAGA). Geneva is a portfolio management system that is used by institutions involved in the trading of investments.

[0025] FIG. 1 shows a hardware and software schema 100 according to an embodiment of the present invention. The schema 100 includes a main memory 110 and secondary storage 120. Resident in the main memory 110 is an application program 130, an operating system 140, and a fault handler 150. The secondary storage 120 includes a data repository 160.

[0026] The application program 130 is programmed to access the data repository 160. The fault handler 150 is associated with the data repository 160 and registered with the operating system 140. In an embodiment, the fault handler 150 is not native to the operating system 140, which may include its own fault handlers. Instead, the fault handler 150 is written particularly to enable interactions between the application program 130 and the data repository 160.

[0027] In an embodiment, the data repository 160 includes various file segments. At any one time, some file segments are mapped into the main memory 110, and other segments are not.

[0028] In an embodiment, when the application program 130 references an object that resides in the data repository 160, but is not currently mapped into the main memory 110, a segmentation fault is issued by the computer hardware at an interrupt location in the application program 130. The fault handler 150 is able to catch the segmentation fault. The fault handler 150 then finds a file segment of the data repository 160 that corresponds to the referenced

object. That file segment is mapped into the main memory 110, and the application program 130 is restarted at the interrupt location.

[0029] In an embodiment, various software components of the schema 100 may be written in an object-oriented programming language, such as C++ or another such language.

[0030] FIG. 2 illustrates a system 200 according to an embodiment of the present invention. The system 200 is an example hardware implementation of the schema 100 of FIG. 1.

[0031] The system 200 includes an application and database server 210. The server 210 includes the secondary storage 120, and may provide a platform for the application program 130, the operating system 140, and the fault handler 150 (not shown) of the schema 100 in FIG. 1. It is to be appreciated that the system 200 may be implemented on one or multiple computers and/or storage devices.

[0032] In an embodiment, the system 200 runs on Sun UltraSPARC or Fujitsu computer systems running Solaris 8 or 9. In general, embodiments herein may be implemented on computer hardware/software systems that support virtual memory, allow user programs to catch segmentation violations, and allow catching routines to restart a faulting application by retrying the instruction that caused the segmentation violation. For instance, embodiments may involve POSIX-compliant systems, such as all varieties of Linux systems, Apple's MacOS X, Sun's Solaris, and Microsoft NT, and its derivatives such as Windows 2000 and XP. In addition, the computer hardware of the system 200 may support 64-bit addressing, with at least 40 bits actually supported by the memory-mapping unit of that hardware. Accordingly, the system 200 may directly access 1 terabyte of data. The larger the number of bits actually supported by the memory-mapping unit, the greater the size of the supported database. Sun SPARC systems, for example, support a 44-bit memory mapping unit, which means that such systems can provide immediate access to 16 terabytes of data. In an example implementation, the computer I/O

system of the system 200 can provide at least 3 megabytes/second of data transfer.

[0033] Returning to the schema 100 of FIG. 1, in an embodiment, data is stored in the data repository 160 in the form of interconnected C++ objects. The objects can be accessed directly by an executing C++ program (e.g., the application program 130) and used as if they were part of the program's local memory. All the stored objects are directly interconnected by memory pointers into one large matrix of information. Only rarely is information searched for as in the classic database model, since almost all information is already pre-linked in the patterns in which it will be used. Unlike relational databases, which may use redundant data tables to represent these pre-linkages, any given data object in the data repository 160 is stored only once. This greatly reduces the total amount of storage required, eliminates database internal consistency problems, and simplifies software development.

[0034] In an embodiment, each object in the data repository 160 has knowledge times (time stamps) associated therewith, indicating when the object was first entered in the database and when it became invalid. Data may become invalid when it is deleted (expired) or updated (superseded by a new variant). This temporal information allows the user to effectively move in time, permitting the reproduction of reports as they would have appeared at any time in the past. In an embodiment, each object has a header defining such knowledge times.

[0035] The application program 130 may attach to the in-memory data repository 160 and map that repository into the virtual memory space of the application program 130. It then accesses the repository objects as if all of them were part of its own memory. The repository objects need not be copied before being given to the application program 130 since they are protected from alteration by memory hardware. An unlimited number of copies of the application program 130 can attach to this shared memory simultaneously. Only one copy can write at any one instant.

[0036] Inside each object is a virtual function pointer that points to a shared memory

8

area that holds the virtual function tables, including virtual functions associated with object types. This pointer technique allows a data repository object to work transparently for any application that touches it. When an application attaches to the data repository 160, a startup routine copies the virtual function table from the application to a specific address in this shared memory, based on an ObjectType field that is stored in each object. Each object in the data repository 160 had its virtual function pointer altered to point to this specific address when it was placed into the data repository 160. Accordingly, each object will now automatically find the correct virtual function definitions for the application that is using it, even if they have changed from the time when the object was originally placed in the knowledgebase.

[0037] Each object also may have a pointer to itself. This allows an object to be asked for its shared memory address, no matter if the object is already in shared memory or it is a local copy. The code need not worry about the actual residency of the object because it will always get a consistent answer.

[0038] Objects can be associated with each other by links. In an implementation, there are three types of linkages in the data repository 160. Y Nodes define the start of like types of objects; X Nodes connect to particular object instances; and Z Nodes are implicit in the objects themselves, pointing from one variant of an object to the next. (Y Nodes actually contain the first X Node as part of themselves. They are shown separately below to more clearly reveal the underlying paradigm.) Linkages may come in a number of variations: zero-or-once, once, once-or-many, zero-or-once-or-many. For example, in a portfolio management embodiment, a Buy can buy one and only one Investment. The link between a Buy and an Investment would therefore be of type "once". Linkage variation rules are enforced at the time that objects or links are placed into the data repository 160.

[0039] In another example, the Buy of a stock may be made in terms of US Dollars (USD). To represent this relationship, the Buy object is linked to the MediumOfExchange

9

invention. The structure 300 may represent how various different objects are linked together in the data repository 160 by the application program 130. The structure 300 is not comprehensive and is merely illustrative of an example structure in a portfolio management database.

[0042] Y Nodes 310 are shown as triangles, X Nodes 320 as circles with X's in them, and Z nodes 330 are represented by variants stacked vertically, such as Deposit 103. The gray objects are in specific Portfolio repository memory segments, and non-gray objects are in the default segments (described below).

[0043] Examples of types of inter- and intra-object pointers are shown in FIG. 3. Since all the objects are linked directly by memory pointers, an application such as the application program 130 can navigate from one data object to another at full memory speed. No "database" operations are required.

[0044] A single object may have dozens of linkages to other repository objects. In an embodiment, since these linkages would quickly come to dominate the storage space, objects that are linked "once" to another object, with no variations in the link, point to a special X Node, called a "unique" X Node. There is one "unique" X Node for each object linked to the main knowledgebase object. This may be especially valuable in an example investments setting which has six different pointers to a MediumOfExchange. All of these pointers are generally invariant, and all normally point to the same object. These pointers are PriceDenomination, BifurcationCurrency, RiskCurrency, IncomeCurrency, PrincipalCurrency, and PriceCrossingCurrency.

[0045] FIG. 4 shows a segmented data repository 400 according to an embodiment of the present invention. The data repository 400 is a logical representation, showing example kinds of data segments in a portfolio management application.

[0046] In an example embodiment, there are five types of data segments in the data repository 400: database, default (or core), portfolio, price, and control. The database segment

11

410 holds those objects that define the database. This segment includes the database logfile write buffer, the current database status, and Segment and SegmentFile objects that point to all the other segments in the data repository 400.

[0047] The price segments 440 contain all non-MediumOfExchange PriceDay objects as well as links to them. Each price segment 440 represents one month of prices for all investments in the associated portfolio management application. The price segments 440 appear as files to the system, with names containing the year and month in human-readable format.

[0048] The portfolio segments 430 hold all the transactions for individual portfolios as well as the links pointing to them, all objects owned by them, and all links pointing to those owned objects. (For example, Reorganization transactions own ReorganizationElements. These ReorganizationElements and the links to their Reorganization parents are all in the same segment as the Reorganization that owns them.) In an embodiment, if the objects are linked to the main knowledgebase object, those links are not placed in the portfolio segments. The Portfolio objects themselves are also not placed in the segments so that they can be searched without paging through the portfolio segments.

[0049] The control segment 450 stores all the UserSession and Agent objects that track usage of the knowledgebase. There is only one control segment 450, just as there is only one database segment 410.

[0050] The default (or core) segment 420 holds everything that is not placed in any other segment. In an embodiment, the default segment 420 holds about 10-20% of the data.

[0051] In an embodiment, a hash table (not shown) resides in the default segment 420. This table allows rapid object access given either primary or secondary keys. Not all object types have entries in this table. Only those that potentially are numerous and might be searched for by key are indexed here. For example, users may look for a particular PortfolioEvent by using a secondary key that they have provided. This table will immediately locate the matching

12

event. The table also may be used to ensure that all primary and secondary keys are unique when a new object is entered into the knowledgebase.

[0052] In an embodiment, objects stored in the memory-mapped file segments of the data repository 160 (FIG. 1) or 400 (FIG. 4) are divided into groups, called species. Example species include Prices, PortfolioEvents, control objects (Agents, UserSessions), derived numerical results (such as Time Weighted Return (TWR) values), and core objects (everything else. An individual segment only contains objects of a particular single species. While the species define the segmentation scheme, an individual within a species may be referred to as a specimen. For example, each Portfolio constitutes a specimen of the PortfolioEvent species. Each PriceMonth constitutes a specimen of the Price species.

[0053] In a particular embodiment, memory-mapped file segments range from 1 to 16 megabytes in size. Segments may grow automatically from minimum to maximum size as objects are added to them, overflowing into new segments if 16 megabytes is insufficient.

[0054] In an embodiment, a user-specified maximum number of segments from each species are held in memory. These segments are evicted from memory on a least-recently-used (LRU) basis. Segments are placed in memory whenever objects that they contain are referenced by the application program 130. The system may run with as little as one segment from any species in memory. As such, a user has essentially total freedom in defining the number of segments that may be concurrently mapped at any one moment.

[0055] In an embodiment, to support the splitting of a data repository into segments, object insertion routines test virtual functions that specify how each object type is to be handled during insert. For example, Portfolio-related events may be stored in clusters that are mapped together in memory based on their associated Portfolio.

[0056] In an example implementation, when a portfolio is added to the data repository, it is assigned a 16-megabyte address at which to start storing its events. This address is a direct

function of the segment identifier that is placed in the Portfolio object. All events associated with this Portfolio will be placed in this allocated memory. Assuming a 44-bit virtual address space, such as provided by SPARC CPUs, more than 1,000,000 Portfolios are supported, each holding about 40,000 events. It is to be understood that reducing the 16-megabyte default size for a segment increases the maximum number of Portfolios that can be supported. In a setting that hosts tens of millions of small, relatively inactive Portfolios, such a reduction may be particularly valuable.

[0057] If the 16 megabyte area reserved for the Portfolio is filled, a new, not necessarily contiguous, allocation is created, and filling of the allocated space resumes. In this way, there is no limit to the size of the stored Portfolio. Segment memory is not completely zeroed when it is allocated; thus, no page faults occur in the unused memory.

[0058] In an embodiment, a segment address allocation algorithm may involve a highest segment address. The highest segment address may be stored in a database object as a reference. When a new segment is required, it is allocated from this address, and the address is then incremented by 16 megabytes.

[0059] When an application process attempts to access memory associated with a Portfolio, memory that is not already mapped will cause a segmentation violation (SIGSEGV). The fault handler then determines if this is a true memory access error or just a request for a segment that is not yet in memory. If the SIGSEGV results from a segment request, the handler memory-maps the segment and restarts the operation.

[0060] In an embodiment, although memory space is allocated in 16-megabyte segments, the underlying mapped files may be created and extended in smaller segments, such as 1-megabyte segments. Such a partial allocation approach may greatly lessen the physical disk space needed to store thousands of small Portfolios and reduces backup and file transfer times.

[0061] Processes detach the segments that they are no longer using. A maximum

14

memory usage may be enforced where segments are unmapped in a least-recently-used (LRU) manner whenever a user-specified limit is reached. In a portfolio management embodiment, only a few months of prices may need to be mapped into memory at any given time.

[0062] In an example implementation, the data repository 160 or 400 holding mapped data segments is stored on a disk subsystem that is connected to a NFS (Network File System) or similar network. Accordingly, the mapped files of the data repository are accessible via NFS from multiple remote computers simultaneously. As such, users who have numerous small computers can team the computers to satisfy large batch processing requirements. Such remote processing is further facilitated by the fact that the network need only transport those data segments that are needed by the remote computers. Such an implementation is scalable, enabling databases to grow extremely large, not limited by hardware memory constraints and associated cost factors.

[0063] It is to be appreciated that, because users can leverage existing networks of computers to accelerate batch runs, TCO (Total Cost of Ownership) is lowered, and batch cycle completion times are improved. In addition, troubleshooting of database problems may be performed more rapidly and responsively, as less data needs to be transferred, and tests may be performed using smaller, more readily available computers.

[0064] FIG. 5 shows a representation 500 of memory-mapped segment files according to an embodiment of the present invention.

[0065] In an embodiment, segment files are named such that they can be quickly located and mapped back into main memory when a corresponding object referenced by an application leads to a segmentation fault. In particular, the names of segment files may relate to the address of the corresponding object that leads to the segmentation fault.

[0066] In an embodiment, the organization of data into memory-mapped segment files is influenced by a consideration of a logical view of the data, such as interrelationships among

15

data. For instance, related data may be clustered together. Accordingly, the amount of data that needs to be mapped into main memory at any one moment may be greatly reduced. In addition, the application program may run faster because cache hit rates may be improved, and TLB (translation lookaside buffer) misses minimized. Further, segment files can be dropped to purge data from the data repository when necessary or desired.

[0067] Since segment files are used to store data, there may be potentially a large number of files stored in segment directories. In an implementation, these files are protected and stored on a device that provides adequate data velocity and capacity. The embodiments herein reduce the amount of swap disk space required to run an application program. This reduction occurs since a multiprocessing operating system must reserve disk swap space equal to the size of the programs kept in process memory. It must reserve this space so that it can move the task out of main memory and onto its swap disk if a higher priority program needs to run. The embodiments herein reduce the amount of swap space that is required, as most of the data is not mapped into memory at any given moment, and that which is mapped into memory is mirrored by the disk files themselves. This means that the operating system does not need to reserve swap disk space for this data, whether it is mapped into memory or not.

[0068] In a particular embodiment, a segment, such as a segment for a Portfolio, may be stored in a Segment Library, which has a two-level directory structure. Two ASCII-formatted, hexadecimal digit sequences, representing a portion of the segment's memory address, create file and directory names. The file name also contains the name of the Portfolio for human accessibility. For example, if a report starts processing the PortfolioEvents for Portfolio Fred, and Fred's events have not previously been used, a memory fault might occur at (hexadecimal) address 0x11234567890. The fault handler for the data repository would then attempt to open the segment directory segment.4635.112, looking for a filename matching the pattern segment.4635.11234.*. The file segment.4635.11234.portfolio.fred.0 will match this pattern,

16

and the fault handler will then map this file at address 0x11234000000 for 16-megabytes. If present, this file is attached and the process is restarted.

[0069] If a very large Portfolio requires more than one segment, its subsequent segments will have ascending final digits; for example, Fred's Portfolio might have files segment.4635.11234.portfolio.fred.0, segment.4635.112f5.portfolio.fred.1, and segment.4635.1134a.portfolio.fred.2. (Segment addresses start at virtual address 0x10000000000, which is 1 terabyte.) It is to be noted that no central lookup table is necessary because the address provides all information that is needed.

[0070] The above naming convention may enable support of multiple data repositories stored in the same directory, as well as access to 15 terabytes out of the 16-terabyte virtual address space. Further, an administrator can easily locate the files belonging to a particular data repository or portion thereof.

[0071] In an example implementation, segmentation also may be employed to store prices. Each PriceMonth, in a main database, points to its child PriceDays, which are stored in their matching segments. When a segmentation violation occurs, the segment is loaded into memory, and processing is resumed. Such operations are transparent from the perspective of the application program.

[0072] Price segments may have names of the form segment.4635.10008.price.200111.0, where 4635 is the hexadecimal data repository name, 10008 indicates that this page maps at address 0x10008000000, price shows that this is a price segment, 200111 indicates that this is a price segment for November, 2001, and 0 indicates that this is the first segment in what might be a chain of segments for this month.

[0073] It is to be appreciated that analogous naming conventions and organizational techniques to those above may be employed in contexts other than portfolio management applications.

17

[0074] FIG. 5 shows example linkages of the stored database objects and the segmentfiles that hold the data that is memory-mapped when referenced. Three segmentfiles are shown. The first is the Database segmentfile 510, which contains the segment 520 and segmentfile 530 objects. The segmentfile 530 objects are normal object segmentfiles. In the example of FIG. 5, they both contain investment price objects for January 1999. The segmentfile names are automatically generated from the keys of the objects being stored and the memory ranges that the data repository routines allocate for them. A segmentfile starts at 1 megabyte in size and can be extended to a maximum of 16 megabytes. If more space is needed, a new segmentfile is created. In various embodiments, a segment may own many non-contiguous segment files.

[0075] FIG. 6 shows a process 600 according to an embodiment of the present invention. The process 600 may be used for memory mapping of databases consistent with the schema 100 of FIG. 1, as well as with other embodiments herein, such as shown in FIGS. 2-5.

[0076] In task 601, an application program connects to a data repository of a database. The data repository includes a plurality of memory-mapped file segments stored on at least one nonvolatile memory medium.

[0077] In task 610, a fault handler for the data repository is registered with the operating system on which the application program runs. In task 620, the fault handler catches a segmentation fault issued for a data repository object that is referenced by the application program but not currently mapped into main memory. The segmentation fault is issued at an interrupt location in the application program.

[0078] In task 630, a file segment of the data repository corresponding to the referenced object is found. In task 640, the found file segment is mapped into main memory. In task 650, the application program is restarted at the interrupt location at which the segmentation fault was issued.

18

[0079] FIGS. 7A and 7B show processes 700, 739 according to embodiments of the present inventions. The processes 700, 739 are similar to the process 600 in some respects. The ellipses in the processes 700, 739 show interfaces to outside routines, such as application program(s). Rounded rectangles show computer hardware actions.

[0080] In task 701 of FIG. 7A, an application connects to the data repository. In task 710, the repository interrupt handler (SEGFAULT) is registered with the operating system. This interrupt handler is able to catch segmentation faults issued by computer hardware during the course of execution of an application.

[0081] In task 720, the application attaches to a license shared memory segment. This task is used to verify that the increase in user count is legal. Task 720 need not be performed in certain embodiments.

[0082] In task 730, all loaded C++ virtual functions are copied or mapped into a special address area.

[0083] In task 733, database and default segment files are mapped into memory at the addresses indicated by their names. For example, the database segment file that is named segment.10000.database.1.0 is mapped into memory starting at location 0x10000000000. Similarly, the first default segment, segment.10001.default.1.0, would be mapped starting at location 0x10001000000. This mapping is done using the same address mapping methodology depicted in FIG. 5.

[0084] In task 736, control is returned to the application.

[0085] Turning to task 740 of FIG. 7B, the application references an object in the data repository. The memory segment for that object may or may not be already mapped into main memory. Task 750 determines which is the case. If the segment is already mapped into main memory, then in task 755, the application continues normally.

[0086] If the segment is not already mapped, then the computer hardware issues a

19

segmentation fault (task 760).

[0087] In task 765, the interrupt handler catches the segmentation fault. It is determined whether the segmentation fault address is in the space controlled by the data repository. If not, then control is returned to the operating system or other fault handlers (task 770).

[0088] If the fault address is in that space, then, if needed, memory constraints are enforced by unmapping another segment (task 775). The disk segment file that represents the address space segment is found, and the file is mapped to main memory (task 785). As mentioned above, segment files may be named in such a way that they can be located quickly based on the address of an object that led to a segmentation fault.

[0089] In task 780, the application is restarted at the exact interrupt location associated with the segmentation fault.

[0090] FIG. 8 shows a process 800 according to an embodiment of the present invention. The process 800 may be applied when new objects need to be stored in the data repository.

[0091] In task 801, an application desiring to add an object to the database calls either an insert or update subroutine as defined by the database API (Application Program Interface). By calling this appropriate subroutine, control is passed to the database routines (task 810), which attempt to perform the corresponding operation and return their success or failure back to the application (task 890).

[0092] In task 820, if the object needs to create a new segment object in the database, the segment object is created, which also loads the object's segmentID. A new segment object, with its new corresponding segmentID, is required if the object being stored is the first instance of a new species member. For example, in an embodiment, suppose a new Portfolio, Jean, is added to the database. When the first trade for Portfolio Jean, a Buy for example, must be inserted, there is no place to put this Buy until a segmentfile is created with a corresponding name and a new segment object to point to it. If the segment object already exists, it will have been

20

retrieved as part of the test performed in task 820 and, as such, the stored object's appropriate segmentID will be known.

[0093] Unless the object specifies its own segmentID or is owned by an object that does, the default segmentID(0) is used (task 830). In task 840, the process determines whether there is space for this object in the current segment file. If so, then space is allocated from the segmentfile, and the address of allocated memory is returned (task 860). If not, the current segment file is extended if possible, and the segment file object is updated (task 850). Otherwise, a new minimum length segment file is created, the segment file object is added to the database, and the segment file object is linked to the segment object (task 870).

[0094] In task 880, the database places the object in the allocated memory. The database returns to the application in task 890.

[0095] Consistent with the above teachings, various example implementations may be realized. In one implementation, support is provided for a 15 terabyte data repository, with up to one million portfolios, using minimal amounts of RAM. Checkpoints lock the entire data repository and perform full file system copies from working to checkpoint directories. There is only one writer for all of the data repository. Multiple-computer support is minimal. Fail-over from one computer to another is not supported.

[0096] In another implementation, the file system copy described above is performed in an unlocked mode, which eliminates checkpoint locking issues. Time stamps at the beginning and end of the file copies allow for backstitching of the log file in such a way that changes are reversed that occurred after the start of the checkpoint. Each file has a header object that records the beginning and ending time of the copy.

[0097] In another implementation, which addresses locking issues, one writer is provided for the control species, and one for all other species. This change relieves conflict between heavy users, such as the loader in all-or-none mode, and users attempting to log onto

the system. Because the control species is locked independently of the others, users can log in while the loader is running; they only need to write to the control species, and the loader never writes to that species. All other transactions gain a lock on all objects except for the control species. Agent and UserSession objects are always written to the control species and are never rolled back. Simultaneous writing is simplified, as the control lock is not held during the entire extent of the lock on all other species.

[0098] Another implementation supports failover or automatic switching from one data repository mother computer to another. This feature provides a user uninterrupted operation when maintenance is required on the normal host computer. Msyncing of memory and baton passing occur between two data repository mother computers. Integrity checking is performed in order to recover information that may not have been fully applied because of a crash of an original host computer that necessitates the switching.

[0099] In yet another implementation, one writing and multiple reading computers operate simultaneously. An existing computer farm may be employed to expedite processing during batch cycles by dividing the work across the machines in the farm.

[00100] The foregoing description of the various embodiments of the present invention is provided to enable any person skilled in the art to make and use the present invention and its embodiments. Various modifications to these embodiments are possible, and the generic principles presented herein may be applied to other embodiments as well.

[00101] For instance, an existing in-memory database may be converted to a memory-mapped database consistent with embodiments of the present invention. Such a conversion may include the provision of secondary storage for a data repository and the programming of modules, such as a fault handler for the data repository.

[00102] It will be apparent to one of ordinary skill in the art that some of the embodiments as described hereinabove may be implemented in many different embodiments of

22

software, firmware, and hardware in the entities illustrated in the figures. The actual software code or specialized control hardware used to implement some of the present embodiments is not limiting of the present invention.

[00103]   Moreover, the processes associated with some of the present embodiments may be executed by programmable equipment, such as computers. Software that may cause programmable equipment to execute the processes may be stored in any storage device, such as, for example, a computer system (non-volatile) memory, an optical disk, magnetic tape, or magnetic disk. Furthermore, some of the processes may be programmed when the computer system is manufactured or via a computer-readable medium at a later date. Such a medium may include any of the forms listed above with respect to storage devices and may further include, for example, a carrier wave modulated, or otherwise manipulated, to convey instructions that can be read, demodulated/decoded and executed by a computer.

[00104]   A "computer" or "computer system" may be, for example, a wireless or wireline variety of a microcomputer, minicomputer, laptop, personal data assistant (PDA), wireless e-mail device (e.g., BlackBerry), cellular phone, pager, processor, or any other programmable device, which devices may be capable of configuration for transmitting and receiving data over a network. Computer devices disclosed herein can include data buses, as well as memory for storing certain software applications used in obtaining, processing and communicating data. It can be appreciated that such memory can be internal or external. The memory can also include any means for storing software, including a hard disk, an optical disk, floppy disk, ROM (read only memory), RAM (random access memory), PROM (programmable ROM), EEPROM (electrically erasable PROM), and other computer-readable media.